Path Problems in Graphs*

Günter Rote †

April 1989

Abstract

A large variety of problems in computer science can be viewed from a common viewpoint as instances of "algebraic" path problems. Among them are of course path problems in graphs such as the shortest path problem or problems of finding optimal paths with respect to more generally defined objective functions; but also graph problems whose formulations do not directly involve the concept of a path, such as finding all bridges and articulation points of a graph; Moreover, there are even problems which seemingly have nothing to do with graphs, such as the solution of systems of linear equations, partial differentiation, or the determination of the regular expression describing the language accepted by a finite automaton.

We describe the relation among these problems and their common algebraic foundation.

We survey algorithms for solving them: vertex elimination algorithms such as Gauß-Jordan elimination; and iterative algorithms such as the "classical" Jacobi and Gauß-Seidel iteration.

Contents

*This work was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 3075 (project ALCOM).

†Institut für Mathematik, Technische Universität Graz, Kopernikusgasse 24, A-8010 Graz, Austria. Electronic mail: `rote@ftug.dnet.tu-graz.ac.at`.
This paper was written while the author was visiting the Institut für Informatik, Freie Universität Berlin, Germany.

1 Introduction

Path problems can be seen as a unified framework for a lot of problems from different fields. Solution procedures for these problems were initially discovered independently of each other. When the connection between these solution methods became apparent, various attempts have been made to lay a common theoretical basis for them. Also, new applications of the method were explored.

It would be difficult to give a complete account of the area of path problems. A complete bibliography including all applications would fill many pages. There have been several good accounts in textbooks and treatises, like Gondran and Minoux [6], chapter 3; Zimmermann [17], chapter 8; Carré [4], chapters 3 and 4.

The purpose of this exposition is to give an introduction to this area and an overview of some of the more interesting applications and interpretations of path problems, and to give a relatively small glimpse of the theory which has been established in this field. We shall do this in a very elementary way.

We shall not deal with specialized algorithms for the shortest path problem in particular. Also, algorithms which use special properties of the underlying graphs will only be mentioned.

The reader who wants to know more about path problems in general or about specific applications should consult the above-mentioned references. References to the literature about various applications are almost completely omitted from this survey unless they appeared recently.

2 Two Example Problems

2.1 Example 1: The shortest path problem

2.1.1 Description of the problem — a numerical example

Consider the directed graph shown in figure 1. It has $n = 4$ vertices and ten arcs, which are labeled with weights. A *path* in a graph is a sequence of $l \geq 0$ vertices $(v_0, v_1, \ldots, v_l)$ such that

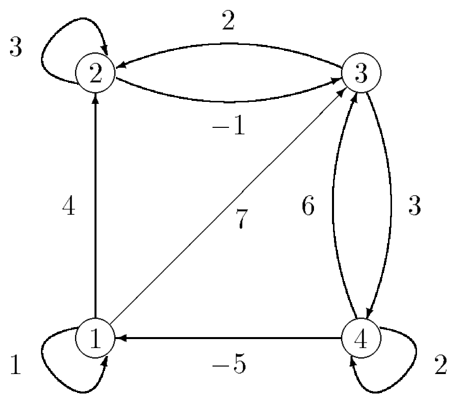

Figure 1: A network

(v_i, v_{i+1}) is an arc of the graph, for $i = 0, 1, \ldots, l-1$. It is called a path *from* v_0 *to* v_j. For example, $p = (1,3,4,4,4,1,3,2)$ is a path from 1 to 2. Note that we allow repetition of vertices and of arcs in a path. With every path, we may associate its *weight*, which is the sum of the weights of its arcs. The weight of the example path p is thus $7+3+2+2+(-5)+7+(-1) = 15$. Note that we must distinguish between an *empty* path without arcs, like the path $q = (1)$ from 1 to 1, and the path $r = (1,1)$, which contains one arc (a loop). The weight of the empty path is assumed to be zero.

The weights can be interpreted as lengths of the arcs, and then the weight of the path is simply its total length. Or the weights could be the time taken to traverse an arc; or the money that one has to pay (or that one gains) for traversing an arc. The last interpretation is one for which arcs of negative weight — as in the example — make sense.

The (all-pairs) shortest path problem is the following:

> For each pair (i, j) of vertices, compute the weight x_{ij} of the *shortest path* (i. e., the path of smallest weight) from i to j.

2.1.2 A system of equations

With the graph G, we may associate its weighted adjacency matrix

$$A = \begin{pmatrix} 1 & 4 & 7 & \infty \\ \infty & 3 & 2 & \infty \\ \infty & -1 & \infty & 3 \\ -5 & \infty & 6 & 2 \end{pmatrix}.$$

The element a_{ij} is the weight of the arc (i, j), if this arc exists. Artificial weights of ∞ have been inserted in the places where no arc exists. These artificial arcs will do no harm, because a path using such an arc has weight ∞; thus it will certainly not affect the shortest path.

Now we are going to set up a system of equations which the desired quantities x_{ij} will fulfill. Consider a shortest path p from i to j. If $i \neq j$, this path must contain at least one arc, i. e., it is of the form $(i = v_0, v_1, \ldots, v_l = j)$, with $l \geq 1$. If it is a shortest path, then the subpath $p' = (v_1, v_2, \ldots, v_l = j)$, must be a shortest path from v_1 to j. Thus $x_{ij} = a_{ik} + x_{kj}$, for some $k = v_1$. On the other hand, the expression $a_{ik} + x_{kj}$, for any k, is the length of some path from i to j, namely the path starting with the arc (i, k) and continuing along the shortest path from k to j. Thus, we have

$$x_{ij} = \min_{1 \leq k \leq n} (a_{ik} + x_{kj}), \quad \text{for } i \neq j. \tag{1}$$

For $i = j$, the above considerations apply with one change: The empty path from i to i without arcs is an additional candidate for the shortest path, and thus we have to extend the above equation:

$$x_{jj} = \min \{ \min_{1 \leq k \leq n} (a_{jk} + x_{kj}), \, 0 \}. \tag{1'}$$

In the above example,

$$X = \begin{pmatrix} 0 & 4 & 6 & 9 \\ 0 & 0 & 2 & 5 \\ -2 & -1 & 0 & 3 \\ -5 & -1 & 1 & 0 \end{pmatrix}$$

is the unique solution of this system, and it represents the lengths of the shortest paths.

2.2 Example 2: The language accepted by a finite automaton

A finite automaton is a machine which reads words (sequences of symbols over some alphabet Σ) and either accepts them or rejects them. It can be specified by its *transition diagram*, which is a finite directed graph (see figure 2). The vertices of the graph are the *states* of the automaton. One of the vertices (vertex 1 in our case) is designated as the *start state*, and a subset of the vertices is designated as the *final states*. The arcs are labeled by subsets of letters from Σ. ($\Sigma = \{f, g, h\}$ in our example.) The automaton starts in the designated start state and reads the symbols of an input word one by one. A label z on an arc (i, j) means the following: If the automaton is in state i and the next symbol which it reads is z, it may go to state j. When the automaton is in state i and there is no arc labeled z which leaves i, the automaton cannot continue and stops. When there is at most one choice of an arc for each state and each input letter, the automaton is called a deterministic automaton; otherwise it is a non-deterministic automaton, but this difference does not concern us here.

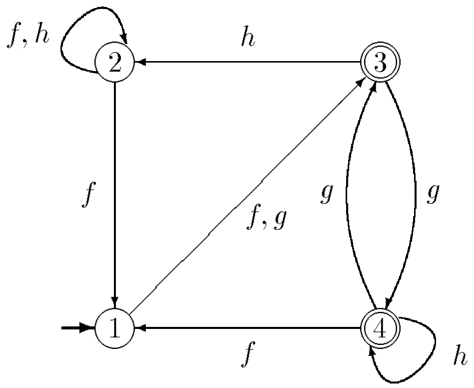

Figure 2: The transition diagram of a finite automaton.
The initial state is state 1. The final states are marked by double circles.

We say that the automaton *accepts* a word, if there is a sequence of state transitions leading from the start state to a final state while reading this word. To put it differently, let $p = (v_0, v_1, \ldots, v_l)$ be a path from the start state v_0 to some final state v_l. If z_i is a label of the edge (v_{i-1}, v_i), for $1 \le i \le l$, then the word $z_1 z_2 \ldots z_n$ is accepted by the automaton. For example, the automaton schown in figure 2 accepts the word *fgghhfhfff* because it leads from state 1 to state 3 via the path $(1, 3, 4, 3, 2, 2, 2, 2, 2, 1, 3)$. Thus, the automaton defines a subset of words (a formal language) which it accepts.

Thus our problem is now the following:

> For each final state j, determine the set x_{1j} of words which lead from the initial state 1 to state j.

In order to solve this problem, we have to introduce a few notations. We are working with words (finite sequences) over some alphabet Σ, including the *empty word* ε, which contains no symbols. We write the concatenation of two words a and b as $a \cdot b$ or simply as ab. If A and B are sets of words, then $A \cdot B$ denotes the set $\{\, ab \mid a \in A, b \in B \,\}$.

As above, we can set up a matrix (a_{ij}), where a_{ij} denotes the set of labels of the arc (i, j). Let x_{ij} denote the set of all words by which the automaton can be lead from state i to state j. We shall solve the more general problem of computing x_{ij} for all pairs of states i and j.

As in the case of the shortest path problem, we shall set up a system of equations. Consider the set x_{ij}. When the automaton is started in state i, the first state transition must lead to some state k. In order to go to k the automaton must read a symbol from a_{ik}. Then it must eventually go to j. The possible words which lead from k to j are collected in the set x_{jk}. Thus, the words which lead from i to j via k as the first vertex are exactly the set $a_{ik} \cdot x_{kj}$. This is also true if there is no arc from i to k, because then $a_{ik} = \emptyset$. Now we just have to take the union over all possible states k, and we get an equation for x_{ij}. Again, if $i = j$, we have to consider the additional possibility that the automaton reads nothing and stays in state i, and thus we have to adjoin the empty word.

$$
\begin{aligned}
x_{ij} &= \bigcup_{k=1}^{n} (a_{ik} \cdot x_{kj}), \quad \text{for } i \neq j \text{, and} \\
x_{jj} &= \bigcup_{k=1}^{n} (a_{jk} \cdot x_{kj}) \cup \{\varepsilon\}
\end{aligned}
\tag{2}
$$

2.3 Summary

In this section, we have described two examples of path problems. In both cases, we have stated the problem, and we have derived a set of equations which the solutions have to fulfill. It is, however, not the case that every solution of the equations is a solution of the respective problem that we started with. We will say more about the relation between the solution of equations and the original formulation of path problems in sections 4.2 and 7.

In the next section we will exhibit the common algebraic structure of our two sample problems.

3 An algebraic framework

3.1 Semirings — the algebraic path problem

The two systems of equations (1)–(1′) and (2) have a similar structure:

$$
\begin{aligned}
x_{ij} &= \bigoplus_{k=1}^{n} (a_{ik} \otimes x_{kj}), \quad \text{for } i \neq j \text{, and} \\
x_{jj} &= \bigoplus_{k=1}^{n} (a_{jk} \otimes x_{kj}) \oplus ①
\end{aligned}
\tag{3}
$$

In the case of the shortest path problem, $\oplus$ denotes max, $\otimes$ denotes $+$, and ① denotes 0, and in the second example problem, $\oplus$ denotes $\cup$, $\otimes$ denotes product (concatenation), and ① means $\{\varepsilon\}$. "$\bigoplus_{k=1}^{n}$" is a notation for the $\oplus$-sum of a sequence of elements, analogous to $\sum_{k=1}^{n}$.

The algebraic structure which is behind these two operations is a *semiring* $(S, \oplus, \otimes)$, i. e., a set S with two binary operations $\oplus$ and $\otimes$, which fulfills the following axioms:

(A_1) $(S, \oplus)$ is a commutative semigroup with neutral element ⓪:

$$
\begin{aligned}
a \oplus b &= b \oplus a, \\
(a \oplus b) \oplus c &= a \oplus (b \oplus c), \\
a \oplus ⓪ &= a.
\end{aligned}
$$

(A_2) $(S, \otimes)$ is a semigroup with neutral element ①, and ⓪ as an absorbing element:

$$\begin{aligned} (a \otimes b) \otimes c &= a \otimes (b \otimes c), \\ a \otimes ① = ① \otimes a &= a, \\ a \otimes ⓪ = ⓪ \otimes a &= ⓪. \end{aligned}$$

(A_3) $\otimes$ is distributive over $\oplus$:

$$\begin{aligned} (a \oplus b) \otimes c &= (a \otimes c) \oplus (b \otimes c), \\ a \otimes (b \oplus c) &= (a \otimes c) \oplus (a \otimes b). \end{aligned}$$

We shall now discuss why these axioms are natural assumptions for any path problem. $\oplus$ must be commutative and associative, because the sum $\bigoplus_{k=1}^{n}$ in equation (3) must be independent of the order of the operands. $\otimes$ is the operation by which the weight of a path is computed from the weights of its arcs, and we require the operation to be associative.

$$w((v_0, v_1, \dots, v_l)) = a_{v_0 v_1} \otimes a_{v_1 v_2} \otimes \dots \otimes a_{v_{l-1} v_l}.$$

① is the weight of the empty path. What we want to compute is, in terms of the semiring, the $\oplus$-sum of the weights of all paths from i to j:

$$x_{ij} = \bigoplus_{\substack{p \text{ is a path} \\ \text{from } i \text{ to } j}} w(p) \tag{4}$$

In this formulation, the problem is called the *algebraic path problem*. However, this formulation contains an infinite sum. This raises questions of "convergence", which fall outside the realm of classical algebra. Thus, we shall mainly stick to the the formulation as a system of equations (3). Later, in section 4.3, we shall also work with the interpretation of x_{ij} as a sum of paths.

We have implicitly used (left) distributivity in the derivation of the equations (1), (2), and (3), when we have expressed the sum of the paths from i to j whose first arc is (i, k) as $a_{ik} \otimes x_{kj}$.

The axioms regarding ⓪ are not essential, since a semiring without ⓪ can always be extended by adding a new zero element according to the axioms, like the element ∞ in the shortest path problem. Thus, we shall not insist that there is always a zero element. (The axioms regarding the existence of ① could also be omitted w. l. o. g., but it requires a trickier construction to show this.) We shall denote the product of an element with itself by the power notation

$$a^k = a \otimes a \otimes \cdots \otimes a \quad (k \text{ times})$$

with the usual convention $a^0 = ①$. Also, for better readability, we shall omit the multiplication sign $\otimes$, from now on.

3.2 Types of semirings, ordered semirings

The examples of semirings which we will encounter belong mostly to three main groups:

1. $(S, \otimes, \leq)$ is a linearly or partially ordered semigroup (with neutral element ①), and $\oplus$ is the supremum or infimum operation (the maximum or minimum operation, in case of a linearly ordered semigroup).

   An ordered semigroup is a semigroup with an order relation which is monotone with respect to the semigroup multiplication:

$$a \leq b \text{ and } a' \leq b' \quad \Longrightarrow \quad a \otimes a' \leq b \otimes b'.$$

When $\oplus$ is defined in this way, it is clearly an associative and commutative operation. The above monotonicity property translates into distributivity. If necessary, we must add a smallest (or largest, resp.) element ⓪.

In the example of shortest paths, the order $\leq$ was just the usual order for real numbers, and $\oplus$ was the minimum operation; in the second example, the order relation is set inclusion, and $\oplus$ is the supremum (least upper bound) with respect to this order.

Another possibility to characterize this class of semirings is that the idempotent law holds for $\oplus$:

$$a \oplus a = a.$$

For this class of *idempotent semirings*, the relation defined by

$$a \leq b \iff a \oplus b = b \tag{5}$$

is a partial order. Thus, we can either start with an ordered semigroup and define $\oplus$ as the supremum operation (if the supremum exists always), or we can start with an idempotent semiring and define the partial order by (5). In both cases we get the same kind of algebraic structure.

2. $(S, \oplus, \otimes)$ is a ring, or a subset of a ring. Examples are the field of real numbers $(\mathbb{R}, +, \cdot)$ with ordinary addition and multiplication, or any subsemiring of the reals, like the natural numbers. For these cases, equation (3) has a closer connection to conventional linear algebra.

3. The elements of S are sets of paths, of path weights, or the like. An example which we have already encounter is the set of words which leads a finite automaton from one state to another. Here, what we deal with are not sets of paths, but sets of label sequences that correspond to paths. Usually, $\oplus$ is the union operation, and thus these semirings fall also under the first category, since they are ordered by the set inclusion relation.

A semiring $(S, \oplus, \otimes)$ with a partial order relation $\leq$ which is monotone with respect to both operations is called an *ordered semiring* $(S, \oplus, \otimes, \leq)$:

$$a \leq b \text{ and } a' \leq b' \quad \Longrightarrow \quad a \oplus a' \leq b \oplus b' \text{ and } a \otimes a' \leq b \otimes b'.$$

All semirings of the first type are ordered semirings, but there are also several examples from the second class, like the non-negative reals $(\mathbb{R}_+, +, \cdot, \leq)$ with the usual order.

We say that $(S, \oplus, \otimes, \leq)$ is ordered by the *difference relation*, or *naturally ordered*, if

$$\text{for all } a, b \in S\text{: } (a \leq b \iff \text{ there is a } z \in S \text{ such that } a \oplus z = b). \tag{6}$$

When $\oplus$ is the min or inf operation of an ordered semigroup, the relation $\leq$ must simply be reversed in order that this definition makes sense. With this proviso, all natural examples of ordered semirings that arise in applications are ordered by the difference relation.

3.3 Matrices

The $(n \times n)$-matrices over a semiring S form another semiring if matrix addition and matrix multiplication are defined just as usual in linear algebra: If $A = (a_{ij})$ and $B = (b_{ij})$ then $A \oplus B = C$ and $A \otimes B = D$, where

$$c_{ij} = a_{ij} \oplus b_{ij}$$

and

$$d_{ij} = \bigoplus_{k=1}^{n} a_{ik} b_{kj}.$$

$(S^{n\times n}, \oplus, \otimes)$ is a semiring. The zero matrix is the matrix whose entries are all ⓪, and the unity element is the unit matrix I with ①'s on the main diagonal and ⓪'s otherwise.

Thus, we may rewrite equation (3) in matrix form as follows:

$$X = I \oplus AX. \tag{7}$$

A symmetric variation of this equation can also be derived by splitting the possible paths from i to j according to their last arc:

$$X = I \oplus XA. \tag{7'}$$

4 Direct solution procedures (elimination algorithms)

We are looking for a solution to the matrix equation (7). If we hope to find a solution for $(n \times n)$-matrices we must surely be able to solve the case of (1×1)-matrices, i. e., of scalars. Thus, we look at the following equation, the so-called *iteration* equation:

$$x = ① \oplus ax. \tag{8}$$

Let us consider what this equation amounts to in the two examples that we have dealt with in the beginning.

$$x = \min\{0, a + x\}$$

For $a > 0$, there is a unique solution $x = 0$. For $a = 0$, the solution of this equation is not unique: any $x \le 0$ is a solution. For $a < 0$, there is no solution.

Correspondingly, the system of equations (1)–(1′) need not have a unique solution, or it can have no solution at all. For example, if we add an arc $(2,4)$ of length 1, then the column vector $(x_{13}, x_{23}, x_{33}, x_{43})$ of the matrix X can be changed to $(-100, -104, -102, -105)$, and we still get a solution. It can be shown that this ambiguity of the solution occurs exactly if the graph contains a cycle of weight 0. In our case, this is the cycle $(1,2,4,1)$.

If we reduce the length of the new arc $(2,4)$ to 0, then no solution fulfills (1)–(1′). The reason is that the graph contains a cycle of negative weight, and hence the shortest paths are undefined. We can remedy this situation by adding a new element $-\infty$ to the semiring. This element solves (8) for $a < 0$. The result $x_{ij} = -\infty$ means then that there are arbitrarily short paths from i to j.

In the semiring of formal languages, we get

$$x = \{\varepsilon\} \cup a \cdot x.$$

This equation always has a solution, namely the set

$$a^* = \{\varepsilon\} \cup a \cup a^2 \cup a^3 \cup \cdots,$$

which consists of all words which are concatenations $w_1 w_2 \cdots w_l$ of an arbitrary number of words $w_i \in a$.

In general, we denote the solution (or some solution) of (8) by a^*, and correspondingly, we denote the solution of the matrix equation (7) by A^*. Semirings in which a^* always exists are called *closed* semirings.

If we repeatedly substitute the expression for x in (8) into the right-hand side, starting with (8), we get

$$\begin{aligned} x &= ① \oplus ax \\ &= ① \oplus a(① \oplus ax) = \cdots = ① \oplus a \oplus a^2 \oplus a^3 \oplus a^4 \oplus \cdots \end{aligned}$$

If this sequence remains stable after a finite numer of iterations, then the sum is a solution of (8).

By multiplying (8) from the right side with any element $b \in S$, we obtain that if $x = a^*$ solves (8) then $y = a^*b$ is a solution of the more general equation

$$y = b \oplus ay. \tag{9}$$

4.1 An elimination procedure — Gauß-Jordan elimination

In this section we shall derive a solution of (3) or (7) by purely algebraic means, namely by successive elimination of variables, very much like in solving ordinary systems of linear equations. Since the intuition for what is acually going on during the solution process may get lost when we write the procedure in full generality, we will first illustrate the method with a specific example. Later, in section 4.3, we will see that the coefficients that arise in the elimination process can be interpreted in a different way, namely as sums of certain subsets of path weights.

When we look at equation (3), we can see that the column index j of the unknowns x_{ij} is the same for all variables which occur in one equation. This means that the system (3) consists really of four decoupled systems of equations, one for each column of X. A column of j respresents the sums of paths from all vertices to the vertex j. Similarly, an equation system for a row of X, i. e., for the paths starting from a fixed vertex i (the *single-source* path problem), can be obtained from (7′).

Let us take a closer look at one specific system, say, for the third column of a (4×4)-matrix:

$$\begin{aligned} x_{13} &= a_{11}x_{13} \oplus a_{12}x_{23} \oplus a_{13}x_{33} \oplus a_{14}x_{43} \\ x_{23} &= a_{21}x_{13} \oplus a_{22}x_{23} \oplus a_{23}x_{33} \oplus a_{24}x_{43} \\ x_{33} &= a_{31}x_{13} \oplus a_{32}x_{23} \oplus a_{33}x_{33} \oplus a_{34}x_{43} \oplus ① \\ x_{43} &= a_{41}x_{13} \oplus a_{42}x_{23} \oplus a_{43}x_{33} \oplus a_{44}x_{43} \end{aligned} \tag{10.0}$$

The quantities a_{ij} are the given data, and the x_{i3} are the unknowns. This is very much like an ordinary system of equations, except that the unknowns appear on both sides: They appear explicitly on the left side, and implicitly on the right side. The iteration equation (9) is the paradigm for handling this situation in the case of one variable: Note that the first equation has the structure

$$x_{13} = ax_{13} \oplus b,$$

with $a = a_{11}$ and $b = a_{12}x_{23} \oplus a_{13}x_{33} \oplus a_{14}x_{43}$. If we assume that a^* exists, then we know that $x_{13} = a^*b$ is a solution of the above equation, and thus we get an explicit expression for x_{13}:

$$\begin{aligned} x_{13} &= a_{11}^* (a_{12}x_{23} \oplus a_{13}x_{33} \oplus a_{14}x_{43}) \\ &= a_{11}^*a_{12}\, x_{23} \oplus a_{11}^*a_{13}\, x_{33} \oplus a_{11}^*a_{14}\, x_{43} \end{aligned}$$

Substituting this into the other equations and collecting terms, we get a new system:

$$\begin{aligned} x_{13} &= a_{12}^{(1)}x_{23} \oplus a_{13}^{(1)}x_{33} \oplus a_{14}^{(1)}x_{43} \\ x_{23} &= a_{22}^{(1)}x_{23} \oplus a_{23}^{(1)}x_{33} \oplus a_{24}^{(1)}x_{43} \\ x_{33} &= a_{32}^{(1)}x_{23} \oplus a_{33}^{(1)}x_{33} \oplus a_{34}^{(1)}x_{43} \oplus ① \\ x_{43} &= a_{42}^{(1)}x_{23} \oplus a_{43}^{(1)}x_{33} \oplus a_{44}^{(1)}x_{43}, \end{aligned} \tag{10.1}$$

where the new coefficients $a_{ij}^{(1)}$ are defined as follows:

$$\begin{aligned} a_{1j}^{(1)} &= a_{11}^*a_{1j}, && \text{for } j > 1, \\ a_{ij}^{(1)} &= a_{ij} \oplus a_{i1}a_{11}^*a_{1j}, && \text{for } i \neq 1,\ j > 1. \end{aligned}$$

Let us summarize what we have done in order to eliminate x_{13}: First we have used the equation where x_{13} occurs on both sides for obtaining an explicit expression of x_{13} in terms of the other variables. This was done by solving the iteration equation. Then we have have used this explicit expression for substituting x_{13} in all other places where it occurred.

The four equations of the last system fall in two groups: The first equation is the explicit expression for x_{13}; the remaining three equations form an implicit system for the other three variables x_{23}, x_{33}, and x_{43}, which has the same structure as the original system, but one variable less.

Thus we can repeat the elimination process in essentially the same way as we have begun it: We eliminate x_{23} from the second equation, assuming that $(a_{22}^{(1)})^*$ exists, and substitute this into the other three equations. We get a new system (10.2), which looks like (10.1) except that x_{23} does not appear on the right-hand side and the superscripts are ${}^{(2)}$ instead of ${}^{(1)}$. The elimination of x_{33} is a bit different, because of the ① on the right-hand side. We get

$$\begin{aligned} x_{33} &= (a_{33}^{(2)})^* (a_{24}^{(1)} x_{43} \oplus ①) \\ &= (a_{33}^{(2)})^* a_{24}^{(1)} x_{43} \oplus (a_{33}^{(2)})^*. \end{aligned}$$

When we substitute this into the other equations, we get a constant term in all equations:

$$\begin{aligned} x_{13} &= a_{13}^{(3)} \oplus a_{14}^{(3)} x_{43} \\ x_{23} &= a_{23}^{(3)} \oplus a_{24}^{(3)} x_{43} \\ x_{33} &= a_{33}^{(3)} \oplus a_{34}^{(3)} x_{43} \\ x_{43} &= a_{43}^{(3)} \oplus a_{44}^{(3)} x_{43} \end{aligned} \tag{10.3}$$

The new coefficients are determined by the following recursions:

$$\begin{aligned} a_{33}^{(3)} &= (a_{33}^{(2)})^*, && \\ a_{i3}^{(3)} &= a_{i3}^{(2)} (a_{33}^{(2)})^*, && \text{for } i \neq 3, \\ a_{3j}^{(3)} &= (a_{33}^{(2)})^* a_{3j}^{(2)}, && \text{for } j > 3, \\ a_{ij}^{(3)} &= a_{ij}^{(2)} \oplus a_{i3}^{(2)} (a_{33}^{(2)})^* a_{3j}^{(2)}, && \text{for } i \neq 3,\ j > 3. \end{aligned}$$

For reasons which will become clear later, we regard the constant terms as the third column of the coefficient matrix. In the remaining elimination steps (there is only one more to follow), this column will remain, whereas the remaining columns will be successively eliminated.

So we finally eliminate x_{43} from the last equation, and we are left with the explicit solution

$$\begin{aligned} x_{13} &= a_{13}^{(4)} \\ x_{23} &= a_{23}^{(4)} \\ x_{33} &= a_{33}^{(4)} \\ x_{43} &= a_{43}^{(4)} \end{aligned} \tag{10.4}$$

with

$$\begin{aligned} a_{44}^{(4)} &= (a_{44}^{(3)})^*, && \\ a_{i4}^{(4)} &= a_{i4}^{(3)} (a_{44}^{(3)})^*, && \text{for } i \neq 4. \end{aligned}$$

The purpose of this calculation has been to make it clear that the solution of the matrix iteration $X = AX \oplus I$ (equation (7)) can be reduced to n solutions of the scalar iteration $x = ax \oplus ①$ for the *pivot elements* $a = a_{11}, a_{22}^{(1)}, a_{33}^{(2)}, a_{44}^{(3)}$. The remaining steps in the derivation were merely substitutions of variables and applications of the semiring axioms (distributivity, etc.) which pose no problems.

Let us summarize in a general way the equations that we have obtained. In the above example, the column index of the solution was $l = 3$. The index k denotes the step number. We denote the elements of the original coefficient matrix by $a_{ij}^{(0)} = a_{ij}$.

$$\begin{aligned} x_{il} &= \bigoplus_{j=k+1}^{n} a_{ij}^{(k)} x_{jl}, && \text{for } 0 \le k < l, i \ne l, \\ x_{ll} &= \bigoplus_{j=k+1}^{n} a_{lj}^{(k)} x_{jl} \oplus ①, && \text{for } 0 \le k < l, \\ x_{il} &= \bigoplus_{j=k+1}^{n} a_{ij}^{(k)} x_{jl} \oplus a_{il}^{(k)}, && \text{for } l \le k \le n, \end{aligned} \tag{11}$$

The formulas for the coefficients $a_{ij}^{(k)}$ were as follows:

$$\begin{aligned} a_{kk}^{(k)} &= (a_{kk}^{(k-1)})^*, \\ a_{ik}^{(k)} &= a_{ik}^{(k-1)}(a_{kk}^{(k-1)})^*, && \text{for } i \ne k, \\ a_{kj}^{(k)} &= (a_{kk}^{(k-1)})^* a_{kj}^{(k-1)}, && \text{for } j \ne k, \\ a_{ij}^{(k)} &= a_{ij}^{(k-1)} \oplus a_{ik}^{(k-1)}(a_{kk}^{(k-1)})^* a_{kj}^{(k-1)}, && \text{for } i \ne k,\ j \ne k. \end{aligned} \tag{12}$$

When we compute only a column x_{il} of the solution, for fixed l, as in our example, we actually carry out the recursions for $a_{kk}^{(k)}$ and $a_{ik}^{(k)}$ only for $k = l$, and the recursions for $a_{kj}^{(k)}$ and $a_{ij}^{(k)}$ only for $j > k$ and for $j = l < k$. Observe, however, that the above recursions are the same *for all columns* l, as far as they overlap for different columns. Thus, when we want to determine the whole matrix X we get just the above recursions, and the final result is

$$x_{ij} = a_{ij}^{(n)}. \tag{13}$$

We get this by setting $k = n$ in (11), whereas for $k = 0$ we get the original system (3) or (10). The nice thing about all these equations is that they can all be interpreted as equations between sets of paths. We will do this in section 4.3.

We can cast our recursion into an algorithm, in which we can omit the superscripts $^{(k)}$ from the variables. We start with the given array a_{ij} and modify this array step by step until the final solution $a_{ij}^{(n)}$ is obtained. This algorithm corresponds just to the Gauß-Jordan elimination algorithm of ordinary linear algebra, and hence it carries this name.

Gauß-Jordan elimination algorithm for
the solution of the equation $X = AX + I$

---

for k **from** 1 **to** n **do begin**
    (∗ Transformation of the matrix $A^{(k-1)}$ into $A^{(k)}$: ∗)
    $a_{kk} := (a_{kk})^*$;
    for all i **from** 1 **to** n **with** $i \ne k$ **do** $a_{ik} := a_{ik} \otimes a_{kk}$;
    for all i **from** 1 **to** n **with** $i \ne k$ **do**
      for all j **from** 1 **to** n **with** $j \ne k$ **do** $a_{ij} := a_{ij} \oplus a_{ik} \otimes a_{kj}$;
    for all j **from** 1 **to** n **with** $j \ne k$ **do** $a_{kj} := a_{kk} \otimes a_{kj}$;
end;

We get a variation of this algorithm if we do not substitute the explicit value for a variable in the equations preceding the current one, only in the following ones. The resulting system for our example would look as follows:

$$\begin{aligned}
x_{13} &= a_{12}^{(1)} x_{23} \oplus a_{13}^{(1)} x_{33} \oplus a_{14}^{(1)} x_{43} \\
x_{23} &= \phantom{a_{12}^{(1)} x_{23} \oplus {}} a_{23}^{(2)} x_{33} \oplus a_{24}^{(2)} x_{43} \\
x_{33} &= \phantom{a_{12}^{(1)} x_{23} \oplus a_{13}^{(1)} x_{33} \oplus {}} a_{34}^{(3)} x_{43} \oplus a_{33}^{(3)} \\
x_{43} &= \phantom{a_{12}^{(1)} x_{23} \oplus a_{13}^{(1)} x_{33} \oplus a_{34}^{(3)} x_{43} \oplus {}} a_{43}^{(4)}
\end{aligned}$$

The system can now be solved in one backsubstitution pass, starting with the last equation. This method corresponds to Gaußian elimination in ordinary linear algebra.

4.2 Theorems about the solution of the elimination algorithm

We can summarize the results of the preceding section as follows:

Theorem 1 *If, for all pivot elements $a = a_{kk}^{(k-1)}$ of the algorithm, a^* is a solution of $x = ① \oplus ax$, then $A^{(n)}$ is a solution of $X = I \oplus AX$.* ■

Note, however, that the converse of this statement is not true: The Gauß-Jordan elimination algorithm may fail although a solution exists. This situation is known from ordinary matrix inversion. There, one cannot always take the next diagonal element as pivot.

We may ask under what conditions the solution of the matrix equation is unique. The following theorem, which follows readily from the elimination algorithm, gives an answer:

Theorem 2 *If, for each pivot element $a = a_{kk}^{(k-1)}$ in the algorithm, a^*b is the* unique *solution of $x = b \oplus ax$, for all $b \in S$, then $A^{(n)}$ is the unique solution of $X = I \oplus AX$.*

Proof. We have to review how the algorithm obtains the solution (10.4) from the original system (10.0). It does so by a sequence of transformations. In going from (10.k–1) to (10.k), we solve one iteration equation $x_{k3} = a_{kk}^{(k-1)} x_{k3} \oplus b$. Under the assumption of the theorem, the resulting equation $x_{k3} = (a_{kk}^{(k-1)})^* b$ is an implication of the iteration equation. The remaining equations of (10.k) are derived by substitution and rearrangement of terms and are therefore also implied by the given equations.

Thus, the final equations (10.n) are implied by the original system, and therefore they represent the unique solution. ■

Note that we must require uniqueness of the solution of $x = b \oplus ax$ for *all* b, since whenever solve an equation of this form during the elimination process, a is a number that we have computed, whereas b is an expression which still involves other unknowns.

For the case of shortest paths, this means that the solution is unique as long as no $a_{kk}^{(k-1)}$ is 0. On the other hand, we have seen that, when the graph contains cycles of zero length, a solution need not be unique. In those cases, it is nevertheless desirable to get a specific solution. In the case of the shortest path problem, the *greatest* solution is the desired solution, since it can be shown that it actually represents the lengths of shortest paths. Thus, we may ask ourselves whether an analog of the above theorem holds for this case, i. e., whether we actually get the greatest (or smallest) solution of $X = I \oplus AX$, if we make sure that a^*b is always the greatest (or smallest) solution of $x = b \oplus ax$.

The following theorem shows that a somewhat weaker statement is true. We formulate it in terms of the smallest solution. One gets an analogous theorem for largest solutions by substituting “smallest” by “largest” and “$\geq$” by “$\leq$”.

Theorem 3 *Assume that we have an ordered semiring. If, for each pivot element $a = a_{kk}^{(k-1)}$ in the algorithm, $y = a^*b$ is the* smallest *solution of $y \geq b \oplus ay$, for all $b \in S$, then $A^{(n)}$ is the smallest solution of $X = I \oplus AX$.*

Proof. We write $\geq$ instead of $=$ in all given equations (10.0) and in all equations (10.k) that are derived during the elimination process. Then, as in the proof of theorem 2, each derived inequality is an implication of the preceding inequalities: For the solution of the iteration equation, this follows from the assumptions of the theorem; for the substitution of the lower bound for this variable in the other inequalities, this follows from the monotonicity of the $\oplus$ and $\otimes$ operations. Since the final inequalities read $X \geq A^{(n)}$, we have the desired result. ∎

If the semiring is ordered by the difference relation (see (6)) we get a result completely analogous to theorem 2, where the "$\geq$" in the preceding theorem is replaced by "$=$". In fact, the following theorem strengthens theorem 2:

Theorem 4 *Assume that we have a semiring which is ordered by the difference relation. If, for each pivot element $a = a_{kk}^{(k-1)}$ in the algorithm, $x = a^*b$ is the* smallest *solution of $x = b \oplus ax$, for all $b \in S$, then $A^{(n)}$ is the smallest solution of $X = I \oplus AX$.*

Proof. Let a be one of the pivot elements of the algorithm. In order to reduce this theorem to the preceding one, we only have to show that a^*b is the smallest solution of $y \geq b \oplus ay$, for any b.

Let y be a solution of the inequality $y \geq b \oplus ay$. Since the semiring is ordered by the difference relation, we may write

$$y = (b \oplus z) \oplus ay,$$

for some z (see (6)). By the assumption of the theorem, $a^*(b \oplus z)$ is the smallest solution of this equation, and hence

$$y \geq a^*(b \oplus z) \geq a^*b \oplus a^*z \geq a^*b.$$

The last inequality follows again from the definition of the difference relation. ∎

Theorems 3 and 4 answer a question posed by Lehmann [9]. The requirement of theorem 4 that the semiring be ordered by the difference relation cannot be omitted completely, as can be shown by a suitable counter-example.

4.3 An interpretation with sets of paths

In this section, we shall give a different interpretation to the equations derived in section 4.1: We shall interpret the coefficients as sums of path weights. These sums are in general infinite. However, in order to avoid the technicalities which are involved in dealing with infinite sums, we shall take a naive approach and assume that all infinite sums exist. In any case, the following considerations can at least be taken as heuristic support for the equations of section 4.1.

The quantity x_{il} represents the sum of the weights of all paths from i to l. We can partition the set of all paths into disjoint subclasses according to some criterion, e. g., according to the first vertex j on the path whose number is greater than i. The paths in one subclass can be split into two subpaths in a unique way, e. g., at this vertex j. By considering all possibilities how this can be done, we get an expression for x_{il} in terms of certain sums of subpaths.

To be more specific, we define a family of sets of paths as follows: We assume that the vertices are numbered from 1 to n. For $1 \leq i,j \leq n$ and $0 \leq k \leq n$, $P_{ij}^{(k)}$ denotes the set of paths from i to j whose intermediate vertices belong to the set $\{1, 2, \ldots, k\}$. The intermediate vertices of a path $(i = v_0, v_1, \ldots, v_l = j)$ are all vertices except the first and the last one. In the case of the empty path (i) we count i as an intermediate vertex; thus, (i) is contained in $P_{ii}^{(i)}$ but not in $P_{ii}^{(i-1)}$.

We shall give the following interpretation of the coefficients $a_{ij}^{(k)}$ that arise in the elimination algorithm:

$$a_{ij}^{(k)} = \bigoplus_{p \in P_{ij}^{(k)}} w(p).$$

For $k = 0$, we get the initial values $a_{ij}^{(0)} = a_{ij}$, which is correct because $P_{ij}^{(0)}$ contains only the arc (i, j), if it is part of the graph. Starting from $k = 0$, the truth of the above expression for $a_{ij}^{(k)}$ can be verified by induction, using the recursions (12). We start with the simplest formula in (12), $a_{kk}^{(k)} = (a_{kk}^{(k-1)})^*$. A path in $P_{kk}^{(k)}$ must start at k and end at k. In the meantime, it can pass arbitrarily many times through k. When we cut the path into pieces at these intermediate vertices k, we get $l \geq 0$ partial paths which are members of $P_{kk}^{(k-1)}$. The expression $(a_{kk}^{(k-1)})^l$ is the sum of all paths which contain exactly $l+1$ occurrences of k (including the first and the last occurrence). Thus, the expression

$$(a_{kk}^{(k-1)})^* = ① \oplus a_{kk}^{(k-1)} \oplus (a_{kk}^{(k-1)})^2 \oplus \cdots \oplus (a_{kk}^{(k-1)})^l \oplus \cdots$$

accounts for every path in $P_{kk}^{(k)}$ in a unique way. On the other hand, it is easy to see that every path weight contributing to the expression on the right-hand side corresponds to the weight of some path in $P_{kk}^{(k)}$.

Now, let us consider the second equation: $a_{ik}^{(k)} = a_{ik}^{(k-1)}(a_{kk}^{(k-1)})^*$, for $i \neq k$. A path in $P_{ik}^{(k)}$ can be split in a unique way into the initial part from i to the first occurrence of k (k must occur since it is the last vertex) and the remaining part. The first part is in $P_{ik}^{(k-1)}$, and the remaining part is accounted for by $(a_{kk}^{(k-1)})^*$. The third equation follows by a symmetric argument (splitting at the last occurrence of k instead of the first occurrence).

The last case $a_{ij}^{(k)} = a_{ij}^{(k-1)} \oplus a_{ik}^{(k-1)}(a_{kk}^{(k-1)})^* a_{kj}^{(k-1)}$, for $i, j \neq k$, is also straightforward. The difference to the previous case is, that a path in $P_{ij}^{(k)}$ need not go through k at all. This is taken into account by the term $a_{ij}^{(k-1)}$.

Using the previous arguments as inductive steps from k to $k+1$, we finally arrive at $a_{ij}^{(n)} = x_{ij}$, because $P_{ij}^{(n)}$ is the set of all paths from i to j (cf. (13)).

Next, we shall consider the equations (11) containing the "unknowns" x_{il} and the "coefficients" $a_{ij}^{(k)}$. In this context, the difference between coefficients and unknowns is immaterial, since we interpret both as sums of path weights.

Let us interpret the first equation in (11): $x_{il} = \bigoplus_{j=k+1}^{n} a_{ij}^{(k)} x_{jl}$, for $0 \leq k < l$ and $i \neq l$. The left side represents all paths from from i to l, for some $i \neq l$. Such a path must contain at least one intermediate vertex whose number is greater than k, because the last vertex l is greater than k. Let j be the first intermediate vertex along the path which is greater than k, and split the path into two parts at this vertex. The first part of the path from i to k contains no intermediate vertex greater than k, which is reflected in the superscript of the expression $a_{ij}^{(k)}$. The second part of the path can be an arbitrary path from k to l. Thus the product $a_{ij}^{(k)} x_{jl}$ is the sum of the weights of all paths from i to l whose first intermediate vertex which is greater than k is j. The vertex j can be any vertex between $k+1$ and n, and thus every path from i to l is represented in a unique way on the right-hand side.

The second equation in (11) differs from the first one only by the additional ① on the right-hand side, which accounts for the empty path in $P_{ll}^{(n)}$. The third equation: $x_{il} = \bigoplus_{j=k+1}^{n} a_{ij}^{(k)} x_{jl} \oplus a_{il}^{(k)}$, for $k \geq l$, differs from the first one in the additional term $a_{il}^{(k)}$. This term accounts for the fact that a path from i to l need not contain an intermediate vertex j whose number is greater than k: The paths which contain no intermediate vertex greater than k are exactly the paths whose weights sum to $a_{il}^{(k)}$.

4.4 Block decomposition methods

As in the case of real matrices, we can decompose a matrix into blocks and carry out the computations blockwise, as with scalar matrices. For example, when we decompose into 4 blocks, the equation $X = AX \oplus I$ becomes

$$\begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \otimes \begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} \oplus \begin{pmatrix} I_{11} & 0 \\ 0 & I_{22} \end{pmatrix}$$

We assume that all diagonal blocks X_{ii} and A_{ii} are square. I_{11} and I_{22} are unit matrices of the appropriate size.

We can apply the elimination algorithm for this block equation without change. The main difference is that the iteration equation $X = AX \oplus B$ which is used to eliminate a variable X_{ij} from the right-hand side is now itself a matrix equation instead of a scalar equation, and the problem of determining A^* is of the same type as the original problem, but of smaller size, however. This opens the possibility for recursive divide-and-conquer solution strategies.

Let us look at the above decomposition into 2×2 blocks and apply the Gauß-Jordan algorithm for $n = 2$.

$$\begin{array}{ll} 1.\ A_{11}^{(1)} := (A_{11})^*; & 5.\ X_{22} = A_{22}^{(2)} := (A_{22}^{(1)})^*; \\ 2.\ A_{21}^{(1)} := A_{21}A_{11}^{(1)}; & 6.\ X_{12} = A_{12}^{(2)} := A_{12}^{(1)}X_{22}; \\ 3.\ A_{22}^{(1)} := A_{22} \oplus A_{21}^{(1)}A_{12}; & 7.\ X_{11} = A_{11}^{(2)} := A_{11}^{(1)} \oplus X_{12}A_{21}^{(1)}; \\ 4.\ A_{12}^{(1)} := A_{11}^{(1)}A_{12}; & 8.\ X_{21} = A_{21}^{(2)} := X_{22}A_{21}^{(1)}; \end{array} \tag{14}$$

We will consider two opposite possibilities for the partitioning strategy: decomposition into equal-size blocks of size approximately $(n/2) \times (n/2)$, and partitioning into one block of size $(n-1) \times (n-1)$ and a scalar.

For the first choice, the above program shows that a *-operation for $(n \times n)$-matrices can be reduced to six multiplications, two additions and two *-operations on $(n/2) \times (n/2)$. By using the reduction recursively, one can derive the result than an $O(n^c)$-time matrix multiplication algorithm for the semiring, with any fixed exponent $c \geq 2$ leads to an algorithm for computing the *-operation with the same asymptotic time complexity (cf. Aho, Hopcroft, and Ullman [2], section 5.9).

The other possibility, where A_{11} consists of the first $n-1$ rows and columns of A and A_{22} is just the element a_{nn}, corresponds to the escalator method for inverting a matrix, which adds one column and one row at a time until the whole matrix is inverted. A_{21} is the last row and A_{12} is the last column of the matrix. $(A_{11})^*$ is the matrix $(a_{ij}^{(n-1)})_{1\leq i,j\leq n-1}$ whose elements correspond to subsets of paths in the graph with vertex n deleted. If we continue the above decomposition recursively, we get the following algorithm for computing A^*. Since the recursive step, the evaluation of A_{11}^*, comes right at the beginning of the algorithm, it is easy to write the algorithm without recursion. For easier reference, we have numbered the steps as in the algorithm above.

Escalator method for the solution of the equation $X = AX \oplus I$

---

for k **from** 1 **to** n **do begin**
  (* Transformation of the matrix $(b_{ij}) = (a_{ij}^{(k-1)})_{1 \le i,j \le k-1}$ *)
  (* into the matrix $(a_{ij}^{(k)})_{1 \le i,j \le k}$. *)
  1. (* $A_{11}^{(1)}$ is already given. *)
  2. **for** j **from** 1 **to** $k-1$ **do** $b_{kj} := \bigoplus_{i=1}^{k-1} a_{ki} b_{ij}$;
  3. $b_{kk} := a_{kk} \oplus \bigoplus_{l=1}^{k-1} b_{kl} a_{lk}$;
  4. **for** i **from** 1 **to** $k-1$ **do** $b_{ik} := \bigoplus_{j=1}^{k-1} b_{ij} a_{jk}$;
  5. $b_{kk} := (b_{kk})^*$;
  6. **for** i **from** 1 **to** $k-1$ **do** $b_{ik} := b_{ik} b_{kk}$;
  7. **for** i **from** 1 **to** $k-1$ **do**
    for j **from** 1 **to** $k-1$ **do** $b_{ij} := b_{ij} \oplus b_{ik} \otimes b_{kj}$;
  8. **for** j **from** 1 **to** $k-1$ **do** $b_{kj} := b_{kk} b_{kj}$;
end;

Note we have used a different array (b_{ij}) for the result variables, because otherwise the k-th row and column would be overwritten while they are being used in steps 2 and 4. Thus the final result is contained in (b_{ij}), whereas the original data (a_{ij}) remain unchanged.

In the case of the shortest path problem, this algorithm is known as the algorithm of Dantzig. There, steps 6 and 8 can be omitted because b_{kk} is always zero, and step 5 reduces to a sign test. Moreover, since the semiring is idempotent, we do not have to differentiate between the matrices (a_{ij}) and (b_{ij}), because it does not matter if elements of A are overwritten.

The recursions of this algorithm can also be interpreted as equations between sets of paths, like in section 4.3. In order to see this, we have to add the correct superscripts. Expressions without superscripts, like a_{ki} denote the initial values of these variables: $a_{ki} = a_{ki}^{(0)}$. Since we have the explicit superscripts, we write a again instead of b:

2. $a_{kj}^{(k-1)} = \bigoplus_{i=1}^{k-1} a_{ki} a_{ij}^{(k-1)}$, for $j < k$: Since a path in $P_{kj}^{(k-1)}$ must contain at least one arc, we can partition this set of paths according to the first vertex i which comes after the start vertex k.

4. $a_{ik}^{(k-1)} = \bigoplus_{j=1}^{k-1} a_{ij}^{(k-1)} a_{jk}$, for $i < k$: This is symmetric to 2.

3. $a_{k}^{(k-1)} = a_{kk} \oplus \bigoplus_{l=1}^{k-1} a_{kl}^{(k-1)} a_{lk}$: This is similar to the previous case, except that we have to take the single arc (k,k) into account.

The remaining recursions:

5. $a_{kk}^{(k)} = (a_{kk}^{(k-1)})^*$;
6. $a_{ik}^{(k)} = a_{ik}^{(k-1)} a_{kk}^{(k)}$, for $i < k$;
7. $a_{ij}^{(k)} = a_{ij}^{(k-1)} \oplus a_{ik}^{(k)} \otimes a_{kj}^{(k-1)}$, for $i, j < k$; and
8. $a_{kj}^{(k)} = a_{kk}^{(k)} a_{kj}^{(k-1)}$, for $j < k$;

are the same as in Gauß-Jordan elimination.

There is also a three-phase algorithm which is analogous to LU-decomposition of ordinary linear algebra (cf. Rote [13]). The top-down pass computes the matrix $L + U$, where L is a strictly lower triangular matrix defined by $l_{ij} = a_{ij}^{(j)}$, for $i > j$, and $l_{ij} = \textcircled{0}$ for $i \le j$, and U is an upper triangular matrix with $u_{ij} = a_{ij}^{(i-1)}$, for $i \le j$, and $u_{ij} = \textcircled{0}$ for $i > j$. Then L^* and U^* are computed, with $(L^*)_{ij} = a_{ij}^{(i-1)}$, for $i > j$, and $(U^*)_{ij} = a_{ij}^{(j)}$, for $i \le j$. Finally, U^* is multiplied with L^*, yielding the result matrix $X = A^*$. The matrices in this algorithm fulfill the following relations:

$$\begin{aligned} A \oplus LU &= L \oplus U \qquad (LU\text{-decomposition}) \\ U^* L^* &= A^* \end{aligned}$$

All of these equations can be interpreted as path equations as in section 4.3.

4.5 A graphical interpretation of vertex elimination

We can view the elimination of a variable from the right-hand side of the equations as the elimination of the corresponding vertex from the graph. This is shown in figure 3. When a vertex k is removed, we must somehow make up for the paths that have gone lost by this removal. Thus, for each pair consisting of an ingoing arc (i,k) and an outgoing arc (k,j), we add a new short-cut arc (i,j). The weight of this additional arc, which is meant to replace the piece incident to vertex k in every path passing through k, reflects the paths that were lost: $a_{ik}a_{kk}^*a_{kj}$. If the arc (i,j) is already present in the graph, we simply add this expression to its old weight.

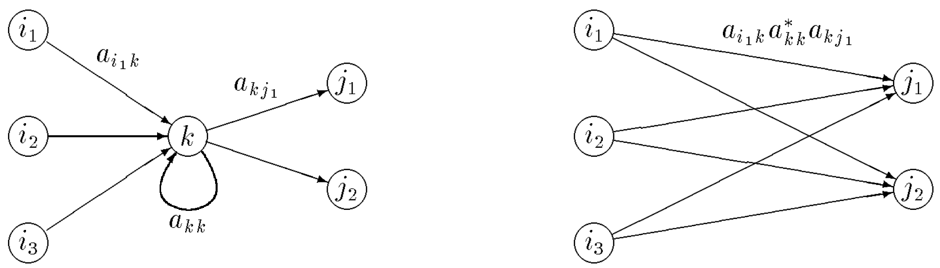

Figure 3: Elimination of the vertex k.

On certain types of sparse graphs (i. e., graphs with few arcs), Gaußian elimination can be carried out more efficiently by using a special ordering in which the variables are eliminated. For example, using a technique called *generalized nested dissection* due to Lipton and Tarjan [11], single-source path problems on *planar* graphs can be solved in $O(n^{3/2})$ steps (see also Lipton, Rose, and Tarjan [10]). Flow graphs from computer programs (cf. section 6.6) usually have a special structure: They are *reducible.* There are specialized algorithms for solving path problems on these graphs (cf. Tarjan [16]).

5 Iterative solution procedures

5.1 Matrix powers

Iterative algorithms are based on the connection between matrix powers and paths of a certain length. In particular, if $(A^l)_{ij}$ denotes the (i,j) entry of the l-th power of the matrix A, then

$$(A^l)_{ij} = \bigoplus_{\substack{p \text{ is a path} \\ \text{from } i \text{ to } j \\ \text{of length } l}} w(p),$$

and thus we get

$$(I \oplus A \oplus A^2 \oplus \cdots \oplus A^l)_{ij} = \bigoplus_{\substack{p \text{ is a path} \\ \text{from } i \text{ to } j \\ \text{of length at most } l}} w(p).$$

For many path problems, paths which are longer than some threshold play no role. For example, in case of the shortest path problem, no path of length n or longer can be a shortest path, and thus it suffices to compute $I \oplus A \oplus A^2 \oplus \cdots \oplus A^{n-1}$. When the semiring is idempotent, such a sum can be evaluated by successively squaring the matrix $(I \oplus A)$. By the idempotence law, we get

$$(I \oplus A)^l = I \oplus A \oplus A^2 \oplus \cdots \oplus A^l.$$

Thus, if we square the matrix $(I \oplus A)$ $\lceil log_2(n-1) \rceil$ times we get a matrix power A^l with $l \geq n-1$, and thus A^l is the matrix of shortest distances.

5.2 Jacobi iteration and Gauß-Seidel iteration

When we want to compute only one row or one column of the matrix X, (i. e., we want to solve the single-source path problem), can simply look at this row of the system (7′) or at this column of the system (7). For column j the corresponding system reads:

$$x = e_j \oplus Ax. \tag{15}$$

e_j demodes the j-th column if I, i. e., the j-th unit vector. One can view this equation, which defines the vector x in terms of an expression involving x, as a recursion which defines a sequence $x^{[0]}, x^{[1]}, x^{[2]}, \ldots$ of successive approximations of x:

$$\begin{aligned} x^{[0]} &:= e_j \\ x^{[l]} &:= e_j \oplus Ax^{[l-1]}, \qquad \text{for } l \geq 1. \end{aligned} \tag{16}$$

Any fixed point of this iteration is a solution of (15). By induction one can show that

$$x^{[l]} = (I \oplus A \oplus A^2 \oplus \cdots \oplus A^l)e_j,$$

i. e., $x^{[l]}$ is the j-th column of the matrix on the right-hand side. In the case of the shortest path problem, this means that the elements of $x^{[l]}$ are the lenghts of shortest paths among the paths which contain at most l arcs. By the results of the previous subsection we conclude that, in case of the shortest path problem, $x^{[n-1]}$ is the j-th column of the shortest path matrix X.

We can simply iterate the recursion (16) until it converges, i. e., until two successive vectors are equal. If the iteration does not converge after n steps, we know that there is a negative cycle. If the iteration converges and the semiring is ordered by the difference relation, the resulting solution is the smallest solution (the *least fixed point*) of the iteration (cf. theorem 4).

This algorithm corresponds to the Jacobi-iteration of numerical linear algebra. Gauß-Seidel iteration is a variation of this method. There, when the elements of the vector $x^{[l]}$ are computed one after the other, they are not computed from the old values of $x^{[l-1]}$, as in (16), but the new elements of $x^{[l]}$ replace the corresponding entries as soon as they are computed. It can be shown that, in the case of idempotent semirings, this modification preserves correctness of the algorithm, and, moreover, the Gauß-Seidel algorithm never needs more iterations than the Jacobi algorithm.

In contrast to elimination algorithms, these iterative algorithms do not require both (left and right) distributive laws. For example, for the column iteration (16) described above, only the left distributive law $a(b \oplus c) = ab \oplus ac$ is required. An example of a semiring where only one of the distributive laws holds occurs in the computation of least-cost paths in networks with losses and gains (cf. Gondran and Minoux [6], section 3.7).

5.3 Acyclic graphs

When the graph $G = (V, A)$ on which we want to solve our path problem is acyclic, one can order the vertices in such a way that an arc (i, j) can only exist if $i < j$. In this case, the matrix entries a_{ij} are zero for $i \geq j$ and the matrix A is (strictly) upper triangular. Then one can solve the system (3) in one pass by computing the solution x_{ij} in the order of decreasing row indices i. Thus, one column of X can be computed in $O(|V| + |A|)$ time. Similarly, one can compute one row of X in linear time by considering the system (7′).

5.4 The Dijkstra algorithm

In some cases, specialized algorithms can solve path problems more efficiently. The single-source shortest path problem in graphs with non-negative arc lengths can be solved efficiently by the algorithm of Dijkstra. This algorithm can be generalized to semirings which come from a linearly ordered semigroup in which ① is the largest element (see the examples in section 6.4.1). The algorithm works by a clever choice of the vertex to be eliminated next. This is somehow analogous to elimination algorithms in linear algebra which use pivoting.

6 Further applications

In this last section, we shall present a selection of examples from different areas which can be interpreted and solved as path problems.

In the first three parts of this section, we shall consider problems which involve the field $(\mathbb{R}, +, \cdot)$ or a subset of it. Then we shall deal with optimization problems; and we shall return to the discussion of finite automata from section 2.2. Finally, we shall present some examples of "non-standard" semirings, which occur in data flow analysis of programs and in two graph-theoretic problems.

6.1 Inversion of matrices

When the semiring is a field, equation (7) can be rewritten as $(I - A)X = I$, or $X = (I - A)^{-1}$ if the matrix $I - A$ is invertible. Then the elimination algorithm corresponds exactly to the Gauß-Jordan algorithm of linear algebra (without pivoting). The only difference is that we get the inverse of $I - A$ and not the inverse of A. This is reflected in the pivoting operation, where we set $a_{kk}^{(k)} := (a_{kk}^{(k)-1})^* = 1/(1 - a_{kk}^{(k)})$ and not $a_{kk}^{(k)} := 1/a_{kk}^{(k)}$.

This problem has originally nothing to do with path problems. We can just pose the equation (7) without reference to a particular graph or to sums of path weights. Nevertheless, the matrix $A^* = (I - A)^{-1}$ has some significance for path problems, as is exemplified in the following two subsections.

6.2 Partial differentiation

Many numerical problems, like finding the minimum of a function f over some domain, can be solved more efficiently if the algorithm has access to the derivative of f. When the function f can be written as a simple expression of one variable, computing the derivative is no problem, but when $f(z_1, \ldots, z_k)$ is a function of several variables, which is computed by a complicated program involving loops and conditional branches, the computation of all partial derivatives $\partial f/\partial z_1$, $\partial f/\partial z_2$, …, $\partial f/\partial z_k$, seems to be a difficult task. Therefore, one used to resort to methods which do not require the derivatives, or they differentiated numerically, which presents new problems of numerical stability.

In this section, we show how the problem of computing the partial derivatives can be solved efficiently as a path problem in a graph, by applying the chain rule.

The graph on which we will work is the *computational graph* of the function f. f is given by a program like the following two-line program, which computes the real root $y = f(z_1, z_2, z_3)$ of the equation $z_1^2 y = z_2 z_3 \sqrt{y} + z_2^2$:

```
y := z2*z3 + SQRT( (z2*z3)^2 + 4*z1^2*z2^2 );
y := ( y / (2*z1^2) )^2;
```

We can resolve this into a sequence of elementary operations, as follows:

1. `a := z2*z3;`	5. `e := c*d;`	9. `y := a+i;`
2. `b := a^2;`	6. `g := 4*e;`	10. `j := 2*c;`
3. `c := z1^2;`	7. `h := b+g;`	11. `k := y/j;`
4. `d := z2^2;`	8. `i := SQRT(h);`	12. `y := k^2;`

Imagine now that this sequence of elementary steps is executed. In the beginning, the graph consists only of k isolated vertices which correspond to the input variables. Each time a variable is assigned a new value, we add a new vertex to the graph, and arcs from this vertex to the one or two operands of this elementary computation (cf. figure 4). When one of the operands is a constant, we first generate a vertex corresponding to this constant. When a variable is assigned several values in succession, we generate different vertices for each assignment (y and $\bar{y}$ in the example).

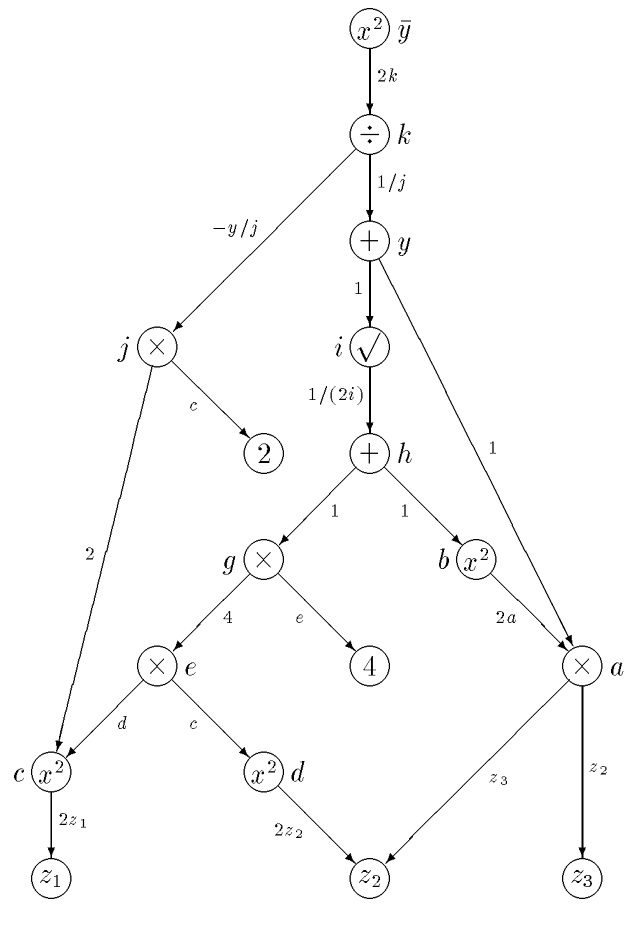

Figure 4: A computational graph

In our case, the computational graph has a static structure, since it corresponds to a straight-line program. However, we can also handle programs with loops and conditional branches, since the computational graph is generated dynamically.

Now let us look at some vertex w with two outgoing arcs leading to vertices u and v. Then we can determine $\partial w/\partial z_i$ by the chain rule:

$$\frac{\partial w}{\partial z_i} = \frac{\partial w}{\partial u} \cdot \frac{\partial u}{\partial z_i} + \frac{\partial w}{\partial v} \cdot \frac{\partial v}{\partial z_i}$$

w is determined from u and v by some elementary operation, and hence $\partial w/\partial u$ and $\partial w/\partial v$ can be calculated directly from u, v, and w in a few elementary basic steps, taking constant time. For example, if $w = u/v$, then $\partial w/\partial u = 1/v$ and $\partial w/\partial v = -u/v^2 = -w/v$. When we associate the value $\partial w/\partial u$ with the arc (w, u) and the value $\partial w/\partial v$ with the arc (w, v), we can write the above equation as follows:

$$x_{wz_i} = a_{wu}x_{uz_i} + a_{wv}x_{vz_i}. \tag{17}$$

Here, a_{wu} and a_{wv} are numbers that can be calculated directly, and x_{wz_i} are the unknowns representing $\partial w/\partial z_i$. In figure 4, the arc weights are shown as small numbers.

We can see that the problem of computing the unknowns is just an instance of the path problem equation (3). In section 2, we have derived (3) starting from a path problem (4). Now arguing in the reverse direction, we obtain:

Theorem 5 *$\partial f/\partial z_i$ is the sum of the weights of all paths from f to z_i in the computational graph, where the weight of a path is the product of its arc weights.*

Since the computational graph is acyclic, we can compute $\partial u/\partial v$, for some fixed vertex u and all other vertices v, or for some fixed vertex v and all other vertices u, in time proportional to the number of arcs of the graph. Since each vertex has at most two outgoing arcs, this is proportional to the number of vertices of the graph, i. e., the number of steps of the algorithm for computing f. Thus, in time which is proportional to the time which the original program takes, we can

- compute $\partial f/\partial z_i$, for all input variables z_i, or
- compute $\partial v/\partial z_i$, for all intermediate variables and output variables v, and for some fixed z_i.

The second problem is solved by a bottom-up pass with a straightforward application of the chain rule (17). This case is interesting if we have a set of functions $f_1(z_1, \ldots, z_k)$, $f_2(z_1, \ldots, z_k)$, $\ldots$, $f_l(z_1, \ldots, z_k)$, with l output variables of the program.

The first problem is solved by a top-down pass through the tree, starting from f. A drawback of this method is that the computational tree must be stored, and hence storage requirement is also proportional to the time complexity of the original program for computing f alone.

Computation of the Jacobi matrix $(\partial f_i/\partial z_j)$ may take much longer than the original program for computing only the l values f_i, since kl values have to be computed. It corresponds roughly to solving the all-pairs path problem.

We can also iterate the procedure for computing derivatives and compute second-order derivatives. Again, note that the time for the computation of the whole Hessian matrix $(\partial^2 f/\partial z_i \partial z_j)$ is also longer than the computation of f, by more than a constant factor, since the Hessian has k^2 entries. However, one can compute the product of the Hessian matrix or the Jacobi matrix with a particular vector in time proportional to the original number of steps of the program.

Note that, in the algorithm, we also determine partial derivatives of f with respect to all intermediate variables. These values can be used to estimate the total rounding error which has been accumulated during the computation of f. For more information, the reader is referred to the survey of Iri and Kubota [8], or to Iri [7], Sawyer [14], or Baur and Strassen [3].

6.3 Markov chains — the number of paths

When the matrix A is the (ordinary) adjacency matrix of a graph, i. e., a_{ij} is 1 if the arc (i,j) exists and 0 otherwise, then the weight of every path is 1, if we use the ring of integers $(\mathbb{Z},+,\cdot)$. Thus, x_{ij} represents the *number* of different paths from i to j. Of course this makes sense only when the graph is acyclic, because otherwise there will be infinitely many paths. On the other hand, from the considerations in section 5.1, we know the the power A^l contains the number of paths of length l between every pair of vertices.

A slight generalization of this is used in the theory of Markov chains. In a Markov chain, there is a finite set $\{1,2,\ldots,n\}$ of states of a system, and the system changes between states in a random way in discrete time intervals. The probability that the system is in state j at some step t depends only on the state of the machine at step $t-1$, and it is independent of t and of previous state transitions. Let a_{ij} be the probability that the system is in state j at step t if it was in state i at step $t-1$. Then the probability that a system passes through a sequence of states $(v_0,v_1,\ldots,v_l)$ is the product $a_{v_0v_1}a_{v_1,v_2}\ldots a_{v_{l-1}v_l}$. Thus the (i,j) entry of the matrix A^l is the probability that the the system is in state j at step t if it was in state i at step $t-l$.

6.4 Optimal paths

6.4.1 Best paths

We have considered the shortest path problem as the first instance of a path problem. There are several other problems, where the set S of path weights is linearly ordered, and the weight of a best path is desired, i. e., $\oplus$ is the operation min or max. Besides the shortest path problem, We have the following examples:

- Maximum capacity paths. The solution uses the semiring $(\mathbb{R}_+ \cup \{\infty\}, \max, \min)$;
- Most reliable paths in networks with possible arc failures, where it is assumed that arc failures of different arcs are independent. Here we use the semiring $([0,1], \max, \cdot)$. The entry a_{ij} of the initial data matrix represents the probability that the arc (i,j) is all right. We are looking for the path with the smallest failure probability.

For the case of maximum-capacity paths, the all-pairs problem can be solved more efficiently by constructing the maximum spanning tree.

The simplest kind of path problem arises when we only ask for the *existence* of a path. Here we take the simplest non-trivial semiring, the Boolean semiring with two elements $(\{0,1\}, \max, \min)$. a_{ij} is 1 if and only if the arc (i,j) exists, and x_{ij} is 1 if j is reachable from i in the graph, i. e., the matrix X represents the transitive closure.

In all cases mentioned above and in the following subsections, the algorithms can be modified such that they will not only compute the weight of an optimal path, but produce the optimal path itself. To achieve this, the algorithms must store how the optimal path weight and each intermediate result was obtained. In some cases, this can only be done at the expense of an increased storage requirement. We will not discuss this in detail.

6.4.2 Multicriteria problems — lexicographic optimal paths

In many applications, paths are not selected according to one criterion, but according to several criteria. In the simplest case, we have a definite order of importance between different criteria. This leads to lexicographic optimization problems.

Imagine that a traveler plans a car trip from one city to another. For each street connecting two points i and j he knows the time t_{ij} to travel from i to j and the amount of sprit s_{ij} that his car needs for this distance. He wants to use as little fuel as possible, but if there are several paths which are equal in this respect, he wants to take the one which takes the shortest time.

Thus, he has a *lexicographic* preference relation $\preccurlyeq$ on the set of vectors (s,t):

$$(s_1,t_1) \preccurlyeq (s_2,t_2) \quad \Longleftrightarrow \quad s_1 < s_2 \text{ or } (s_1 = s_2 \text{ and } t_1 \le t_2).$$

This is a linear order of the vectors $(s,t) \in \mathbb{R}^2_+$. In the semiring, the operation $\oplus$ is the lexicographic minimum, whereas $\otimes$ is the ordinary elementwise vector addition:

$$\begin{aligned}
(s_1,t_1) \oplus (s_2,t_2) &= \begin{cases} (s_1,t_1), & \text{if } s_1 < s_2 \\ (s_2,t_2), & \text{if } s_2 < s_1 \\ (s_1, \min\{t_1,t_2\}), & \text{if } s_1 = s_2. \end{cases} \\
(s_1,t_1) \otimes (s_2,t_2) &= (s_1+s_2, t_1+t_2)
\end{aligned}$$

One can even use a different $\otimes$-operation for the components. For example, imagine that the trip goes through the desert. If several journeys have the same time *and* the same fuel consumption, the traveler wants to select the safest trip among them, i. e., he wants his minimum safety reserve, below which his tank will never be emptied, to be as high as possible. This means that the sprit requirement between any two successive visits to filling stations on his journey should be as low as possible. Thus, we have a different semiring, where $\otimes$ is defined as

$$(s_1,t_1,s_1') \otimes (s_2,t_2,s_2') = (s_1+s_2, t_1+t_2, \max\{s_1',s_2'\}).$$

As before, $\oplus$ is the lexicographic minimum operation (of three components, this time).

Note however, that if the primary goal of our traveler is safety, whereas total fuel consumption and time have second and third priority, the corresponding operations $\otimes$, in which the third component would come in the first place, would not yield a semiring, because it violates the associative law. (The structure $(\mathbb{R}^3_+, \otimes, \preccurlyeq)$ would not be an ordered semigroup.)

If there is no clear preference between the objectives (of fuel over time, or vice versa), we can still eliminate from consideration a path which is worse than some other path is both respects. What remains is the set of *efficient* or *Pareto-optimal* or minim*al* paths; i. e., We are looking for all path weights (s,t), for which there is no other path with weight (s',t') such that $s' \le s$ and $t' < t$, or $s' < s$ and $t' \le t$.

This problem can also be formulated as a path problem, with *sets* of pairs (s,t) as elements of the semiring. The $\otimes$ operation for sets is the elementwise $\otimes$-product of the elements, and $\oplus$ is set union. However, after every operation, we can reduce the resulting sets by throwing away pairs (s,t) which are not efficient. Since these sets of efficient values can become very large, this approach is limited to small problems.

6.4.3 k-best paths

Another extension of the ordinary best path problem is the determination of the k best different paths between every pair of vertices. We can solve this by a semiring which operates on vectors of k elements. For simplicity we will assume that we want to compute the k shortest paths in the ordinary sense, i. e., we are working with the semiring $(\mathbb{R}_\infty, \min, +)$. However, the underlying semiring for the corresponding best path problem can be any semiring in which $\oplus$ is the min or the max operation (cf. the examples in the previous subsections). We are going to create a semiring $(S^k, \oplus, \otimes)$, whose elements are k-tuples of S. The vector $(a_1, a_2, \ldots, a_k)$ is meant to represent the lengths of the best, the second-best, ..., the k-best path in a certain set of paths. The operations for the semiring are defined as follows:

- $(a_1,a_2,\ldots,a_k) \oplus (b_1,b_2,\ldots,b_k)$ is the sequence of the k smallest values in the combination (union) of the two given sequences.

- $(a_1,a_2,\ldots,a_k) \otimes (b_1,b_2,\ldots,b_k)$ is the is the sequence of the k smallest values in the (multi-)set of k^2 elements $\{\, a_i + b_j \mid i = 1,\ldots,k;\ j = 1,\ldots,k \,\}$.

Note that a sequence $(a_1, a_2, \ldots, a_k)$ can contain repeated elements, which correspond to different paths with the same length. Moreover, addition in this semiring is not idempotent. However, the order of the elements in the sequence is immaterial, and thus we might just as well assume that they are sorted. Thus, by the way we treat the sequences in S^k, they are in fact multisets.

- ① $= (0, \infty, \ldots, \infty)$ and ⓪ $= (\infty, \ldots, \infty)$. The initial entries of the matrix are $(a_{ij}, \infty, \ldots, \infty)$, when a_{ij} is the weight of the arc (i, j). (Initially, the path (i, j) is the only path from i to j that we know of. The second-, third-best, etc., paths do not exist.)

- We have to define the *-operation, i. e., the solution of

$$x = (0, \infty, \ldots, \infty) \oplus a \otimes x. \tag{18}$$

  Let us assume that $a = (a_1, a_2, \ldots, a_k)$ with $a_1 \le a_2 \le \cdots \le a_k$. When we write x as ① $\oplus\, a \oplus a^2 \oplus \cdots$, we see the following:

  - If $a_1 < 0$, there is no solution, except perhaps $(-\infty, -\infty, \ldots, -\infty)$.
  - If $a_1 = 0$, we get $x = (0, 0, \ldots, 0)$ as the largest solution. However, any constant vector $x = (c, c, \ldots, c)$ with $c \le 0$ is also a solution of (18). (These are not the only solutions.)
  - If $a_1 > 0$, there is a unique solution, which can be determined by looking at equation (18): $x = (x_1, x_2, \ldots, x_k)$ consists of the k smallest values in the (multi-)set

$$\{0\} \cup \{\, a_i + x_j \mid i = 1, \ldots, k;\ j = 1, \ldots, k\,\}. \tag{19}$$

    We see that the smallest element in this set is $x_1 = 0$, since the elements of the right-hand set of the union are all positive. Let us assume that we have determined the l smallest elements of the set (19). x_{l+1}, the $(l+1)$-smallest element of this set is the l-smallest element of the right-hand set. However, in order to determine the l-smallest element of $a \otimes x$, we need only know the l smallest elements of x, which we know already. Thus, we can successively determine $x_1, x_2, \ldots, x_k$.

Let us discuss the complexity of these operations. $\oplus$ can clearly be carried out in $O(k)$ time. The operation $\otimes$ can also be carried out in (theoretical) $O(k)$ time, using a sophisticated algorithm of Frederickson and Johnson [5]. The determination of a^* as described above, which occurs only n times during the elimination algorithm, can be carried out in $O(k \log k)$ steps, using priority queues.

Note that we do not get the k best *elementary* paths by this approach; i. e., the paths that we get can contain repeated vertices and arcs. The problem of finding the k best elementary paths is considerably more difficult, but also for this problem, algorithms which use the algebraic framework of the path problems have been proposed.

The k-best path problem is an example of a path problem where non-elementary paths can have an influence on the solution. However, in case there are no negative cycles, the longest path that has to be taken into account has $kn - 1$ arcs. (For a longer path, one can construct at least k different shorter paths by successively eliminating elementary cycles from the path.)

Thus, the iterative algorithms of section 5 should converge after at most $kn - 1$ iterations, unless there are negative cycles.

6.5 Regular expressions

For our second example from the beginning, the determination of the language accepted by a finite automaton (cf. section 2.2, the elimination algorithm seems to be useless, since the *-operation will probably very soon lead to infinite sets. However, the algorithms provides us with a way to describe the language. In order to explain this, we need one more definition: A *regular language* is a set of words which is built starting from finite sets of words using only the operations $\cdot$ (concatenation), $\cup$, and *. For example, $e(\{fh^*(f \cup \{g, h\}^*)\}^* \cup \{\varepsilon, ggg\}$ is a regular language. (Here, single words denote singleton sets.) Now the Gauß-Jordan elimination algorithm successively constructs a regular expression for each pair of states i and j, which describes the language x_{ij} leading from i to j: It start from the finite sets a_{ij}, and as it proceeds, it uses only the operations $\cdot$, $\cup$, and *.

Since the language accepted by the automaton is just the union of several x_{ij}, we have proved the following theorem:

Theorem 6 *The language accepted by a finite automaton is regular.* ■

This is one half of Kleene's theorem about the equivalence of finite automata and regular expressions. The other direction, the construction of a finite automaton which accepts a given regular language, is even easier. Our proof by Gauß-Jordan elimination is in fact the standard proof of this result.

6.6 Flow analysis of computer programs

When a compiler wants to optimize the code for a computer program, for example by detecting common subexpressions or by moving invariant expressions out of loops, it needs to know whether the value of an expression remains unchanged between two uses of this expression. If this is the case, the expression need not be evaluated the second time.

In order to investigate this problem, for one particular expression $f(z_1, \ldots, z_k)$ which occurs in the program, we represent the program by its *flow graph.* The vertices correspond to basic blocks of the program, i. e., blocks of consecutive statements with one entry point at the beginning and one exit point at the end. The arcs indicate possible transfers of control between basic blocks. (This is similar to a flowchart.)

The execution of a basic block may have one of the following effects on the value of f:

- It may *generate* f, i. e., the value of f is computed in the block and is available on exit from this block.
- It may *kill* f, for example by assigning a new value to one of the input variables $z_1, \ldots, z_k$ of f.
- It may leave f *unchanged.*

We give an arc (i, j) the label G, K, or U, depending on whether the value of f is generated, killed, or left unchanged between the entry to block i and the entry to block j, (i. e., during the execution of block i). In addition, we need an element ⓪ for the arcs which are not present.

Now we can use the following semiring on the set $\{⓪, G, U, K\}$.

$\oplus$	⓪	G	U	K
⓪	⓪	G	U	K
G	G	G	U	K
U	U	U	U	K
K	K	K	K	K

$\otimes$	⓪	G	U	K
⓪	⓪	⓪	⓪	⓪
G	⓪	G	G	K
U	⓪	G	U	K
K	⓪	G	K	K

a	a^*
⓪	U
G	U
U	U
K	K

U is the ①-element of this semiring. All semiring axioms hold. Note that in this semiring the operation $\otimes$ is not commutative. The operation $\oplus$ is just the min operation for the order $K < U < G <$ ⓪. This is typical of data flow problems, because when we unite two sets of possible paths from i to j, we can only keep the weaker information of the information that the two sets give about f.

To solve our original problem, let 1 be the start vertex of the program. We can eliminate an evaluation of the expression f in block j if and only if x_{1j} is G.

Further examples of data flow problems and references can be found in Tarjan [15].

6.7 Some graph-theoretic problems

The following examples are mentioned mainly as curiosities, in order to illustrate the broad range of applicability of the path problem formulation. For each of the problems, there are in fact linear-time algorithms to solve them directly.

The transitive closure of a graph, which also falls into this category, has already been mentioned briefly in section 6.4.1.

Path problem formulations have also been proposed for problems of enumerating elementary paths or cutsets of a graph. Such problems are exponential by their output size alone. There the solution procedures by elimination algorithms can be applied only to graphs of moderate size.

6.7.1 Testing whether a graph is bipartite

An undirected graph is bipartite if it contains no odd cycle (i. e., no cycle containing an odd number of edges). Since a cycle is just a special case of a path from a vertex to itself, we can formulate this as a path problem. Let the weight of a path be E or O according to whether its length is even or odd, and let the weight of a set of paths be $\emptyset$, E, O, or EO, according to whether the set is empty, contains only even paths, only odd paths, or paths of both types. Then we get the following semiring on the four-element set $S = \{\emptyset, E, O, EO\}$:

$\oplus$	$\emptyset$	E	O	EO
$\emptyset$	$\emptyset$	E	O	EO
E	E	E	EO	EO
O	O	EO	O	EO
EO	EO	EO	EO	EO

$\otimes$	$\emptyset$	E	O	EO
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
E	$\emptyset$	E	O	EO
O	$\emptyset$	O	E	EO
EO	$\emptyset$	EO	EO	EO

a	a^*
$\emptyset$	E
E	E
O	EO
EO	EO

We initialize the data matrix by setting $a_{ij} = O$ if the edge $\{i, j\}$ exists and $a_{ij} = \emptyset$ otherwise. Then, if any $x_{ii} = O$ or EO when the algorithm stops, the graph is not bipartite; otherwise it is. (Of course, as soon as the element EO appears somewhere in the matrix, we know already that the graph is not bipartite.)

We can also apply this algorithm to *directed* graphs and test for the existence of paths of given parity. Using a generalization of this idea, one can find shortest even paths or shortest odd paths, if one does not insist that the paths should be elementary. One could even find a shortest path whose number of arcs is, for example, congruent to 4 modulo 7, if one wishes to do so.

6.7.2 Finding the bridges and the cut vertices of a graph

A *brigde* in an undirected graph $G = (V, E)$ is an edge whose removal causes some connected component of G to break into two components. Similarly, a *cut vertex* or *articulation point* is a vertex whose removal causes some connected component of G to become disconnected.

For finding bridges, we use the semiring $(2^E \cup \{⓪\}, \cap, \cup)$, which operates on the set of subsets of edges augmented by a zero element. $\cap$ and $\cup$ are the ordinary set intersection and set union operations, except that their interaction with ⓪ is specified by the semiring axioms. As unity element we have $① = \emptyset$, and thus the *-operation presents no problem: $x = ① \oplus ax = \emptyset \cap (a \cup x)$ always has the same unique solution $x = \emptyset$, even when $a = ⓪$.

As the weight of an arc (i, j) we take simply the singleton set $\{e\}$, if $e \in E$ is the (undirected) edge corresponding to the (directed) arc (i, j), and we take ⓪ if no such edge exists. The weight of a path is then just the set of its edges. Using the formulation (4) of the algebraic path problem, we get:

$$x_{ij} = \bigcap_{\substack{p \text{ is a path} \\ \text{from } i \text{ to } j}} w(p).$$

In other words, x_{ij} is the set of edges which belong to *every* path from i to j. Such edges are clearly bridges, since their removal causes i to become disconnected from j. Conversely, every bridge must appear in some set x_{ij}.

Note that the semiring axioms would allow us to take the set E as the zero element instead of ⓪. But then we could not distinguish the case where all edges are bridges ($x_{ij} = E$) from the case when i and j are not connected ($x_{ij} =$ ⓪).

The determination of cut vertices proceeds in essentially the same way. We use the semiring ($2^V \cup \{$⓪$\}, \cap, \cup$), and the weight of (i, j) is $\{i\}$, if $\{i, j\} \in E$, and ⓪ otherwise. All elements of the set x_{ij}, except for i, are cut vertices.

7 Conclusion — comparison of different approaches

The general path problem can be approached in several different ways. They are charaterized by different formulations and by different assumptions about the underlying algebraic structure.

We have taken a purely algebraic approach: Solve the system of equations (3).

The usual approach is more direct. It involves the formulation of the problem as an infinite sum (4) of path weights and building up this sum by computing sums $a_{ij}^{(k)}$ of larger and larger path sets, using the equations (12). We have seen this approach in section 4.3. With suitable axiomatic assumptions for infinite sums this derivation of the solution can be made precise. In some semirings infinite sums do not always exist, although they can be defined for *some* sequences. These semirings include the important case of the real numbers $(\mathbb{R}, +, \cdot)$ with their rich structure of convergence. Such semirings can also be dealt with quite satisfactorily. This approach has for example been taken in Rote [13].

A variation of this method specifies the solution in the free semiring generated by the arc set. This is the semiring of multisets of paths with set union and concatenation as addition and multiplication operations. The solution for a specific semiring is then obtained by applying a homomorphism from the free semiring to the specific semiring. An approach like this is taken by Tarjan [15].

Lehmann [9] has taken the 2×2 block decomposition algorithm (14) as the recursive definition of A^* for matrices in terms of the operation a^* for scalars. He shows that the result is independent of how the matrix is decomposed into blocks. A similar approach is taken by Abdali and Saunders [1] who introduce the concept of eliminants to define A^*. Their definition corresponds to a particular way of computing A^* in terms of the a^* operation, very similar to our elimination algorithm.

A comparison of different approaches can be found in Mahr [12].

For some applications, like shortest paths, the formulation (4) involving sums of path weights is more natural, whereas the algebraic formulation (3) is more convenient for other applications such as the inversion of matrices. However, the relationship between the two formulations is not so close: The system (3) may have a solution although the infinite sum (4) makes no sense (consider the case of matrix inversion), or it may have several solutions (cf. the discussion of the shortest path example at the beginning of section 4). However, the desired solution of (3) can often be characterized as the smallest (or largest) solution. Theorems 3 and 4 of section 4.2 show that this desired solution can be obtained by defining a^* appropriately.

We hope that the broad range of applications from which we could draw our examples has convinced the reader of the importance and the general usefulness of path problems.

References

[1] S. K. Abdali and B. D. Saunders, Transitive closure and related semiring properties via eliminants, *Theoret. Comput. Sci.* **40**, (1985), 257–274.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading (Mass.) etc. 1974.

[3] W. Baur and V. Strassen, The complexity of partial derivatives, *Theoret. Comput. Sci.* **22** (1983), 317–330.

[4] B. A. Carré, *Graphs and Networks*, The Clarendon Press, Oxford University Press, Oxford 1979.

[5] G. N. Frederickson and D. B. Johnson, The complexity of selection and ranking in $X + Y$ and matrices with sorted columns, *J. Comput. Syst. Sci.* **24** (1982), 197–208.

[6] M. Gondran and M. Minoux, *Graphes et algorithmes*, Editions Eyrolles, Paris 1979; English translation: *Graphs and Algorithms*, Wiley, Chichester etc. 1984.

[7] M. Iri, Simultaneous computation of functions, partial derivatives, and estimates of rounding errors, *Japan J. Appl. Math.* **1** (1984), 223-252.

[8] M. Iri and K. Kubota, Methods of fast automatic differentiation and applications, Research Memorandum RMI 87-02, University of Tokyo, Faculty of Engineering, Hongo 7-3-1, Bunkyo-ku, Tokyo, July 1987.

[9] D. J. Lehmann, Algebraic structures for transitive closure, *Theoret. Comput. Sci.* **4** (1977), 59–66.

[10] R. J. Lipton, D. J. Rose, and R. E. Tarjan, Generalized nested dissection, *SIAM J. Numer. Anal.* **16** (1979), 346–358.

[11] R. J. Lipton and R. E. Tarjan, A separator theorem for planar graphs, *SIAM J. Appl. Math.* **36** (1979), 177–189.

[12] B. Mahr, Iteration and summability in semirings, in: R. E. Burkard, R. A. Cuninghame-Green, U. Zimmermann (eds.), *Algebraic and Combinatorial Methods in Operations Research* (Proc. Workshop on Algebraic Structures in Operations Research, Bad Honnef, Germany, April 1982), Annals of Discrete Mathematics **19**, North-Holland, Amsterdam 1984, pp. 229–256.

[13] G. Rote, A systolic array for the algebraic path problem, *Computing* **34** (1985), 191–219.

[14] J. W. Sawyer, jr., Fast partial differentiation by computer with an application to categorical data analysis, *The American Statistician* **38** (1984), 300–308.

[15] R. E. Tarjan, A unified approach to path problems, *J. Assoc. Comput. Mach.* **28** (1981), 577–593.

[16] R. E. Tarjan, Fast algorithms for solving path problems, *J. Assoc. Comput. Mach.* **28** (1981), 594–614.

[17] U. Zimmermann, *Linear and combinatorial optimization in ordered algebraic structures*, Annals of Discrete Mathematics **10**, North-Holland, Amsterdam 1981.